2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Heterogeneous Algorithmic Skeletons for FastFlow with Seamless Coordination over Hybrid Architectures

Mehdi Goli
IDEAS Research Institute,
Robert Gordon University, Aberdeen, UK
Email: m.goli@rgu.ac.uk

Horacio González–Vélez
Cloud Competency Centre,
National College of Ireland, Dublin, Ireland
Email: horacio@ncirl.ie

***Abstract*—Algorithmic skeletons ('skeletons') abstract commonly-used patterns of parallel computation, communication, and interaction. They provide top-down design composition and control inheritance throughout the whole structure. The efficient execution of skeletal applications on a heterogeneous environment has long been of interest to the research community. Arguably, executing a coarse-grained resource-intensive skeletal workloads ought to achieve higher resource utilisation and, ultimately, better job makespan on heterogeneous systems due to the structured parallelism model. This paper presents a heterogeneous OpenCL-based GPU back-end for FastFlow, a widely-used skeletal framework. Our back-end allows the user to easily write any arbitrary OpenCL code inside an heterogeneous algorithmic skeleton and seamlessly control the allocation of OpenCL kernel over the hybrid (CPU/GPU) architecture. Our performance evaluation indicate that a skeletal program which employs our back-end is around one order of magnitude faster than a skeletal parallel program using the traditional homogeneous FastFlow skeletons with the serial version of OpenCL code.**

***Keywords*-Algorithmic Skeletons; Parallel Patterns; Structured Parallelism; GPU; Parallel Programming; OpenCL**

I. Introduction

Efficient heterogeneous parallel computing on CPU and GPU (Graphics Processing Unit) architectures remains an open area of interest in computational science. Despite the presence of high-performance interconnects and algorithmic palliation, the complexity of CPU/GPU architectures commonly induce performance trade-offs associated with the communication and non-linear performance characteristics in GPU-accelerated nodes. With the increased development and installation of major CPU/GPU architectures, enhanced techniques and tools for managing these heterogeneous platforms are of increasing significance.

Specifically designed to support architectural heterogeneity, OpenCL [1] provides a unique framework for both task- and data-parallel programming paradigms for both general- and special-purpose architectures including GPUs. Originally developed by Apple, the OpenCL specification [2] is now managed by the Khronos Group, not-for-profit industry consortium that maintains a number of open standards for parallel computing, media, and signals. The specification has been implemented by different hardware vendors making GPU code portable, and effectively enabling OpenCL code to be run on different cards *without modification*. Furthermore, well-structured OpenCL code ought to fall back to CPU-only execution in order to consider those cases when a host system does not furnish any supported devices. Overall, OpenCL is arguably suitable for hybrid CPU/GPU architectures and cloud-based environments.

Algorithmic skeletons (or simply skeletons) have long been considered as a viable approach to introduce high-level abstraction to parallel programming that hides the complexity of recurring patterns of coordination and communication logic behind a generic reusable application interface [3], [4]. The FastFlow [5] framework is a C++ shared memory algorithmic skeleton implementation developed for cache-coherent multi-core architectures, which has demonstrated better performance than the Intel TBB library and some OpenMP implementations [6]. The FastFlow framework has been extensively tested in shared-memory multi-core environments, however integrated GPU support within the libraries remains under development.

In particular, skeletal parallel pipelines enable the decomposition of a repetitive sequential process into a succession of distinguishable sub-processes called stages, each of which can be *concurrently* executed on a distinct processing element. Coarse-grained parallel pipelines refine complex algorithmic solutions into a sequence of independent computational stages where the data is "piped" from one computational stage to another. When handling a large number of streaming inputs, it is throughput rather than latency which constrains overall efficiency, since the latency is only relevant to measure the time to fill up the pipeline initially. Once at capacity, the pipeline steadily delivers results at the throughput ratio. Hence, in order to improve the overall efficiency of a parallel pipeline, it is necessary to minimise the bottleneck processing time by allocating the distinct stages to the most suitable (CPU/GPU) processing element.

Arguably, any suitable parallel programming environment should not be confined to a given architecture in order to efficiently deploy a parallel application. We therefore hypothesise that, by integrating OpenCL support into FastFlow,

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.29

we can efficiently coordinate a computationally-intensive pipeline application on a tightly-coupled heterogeneous multi-core CPU/GPU cluster architecture and, ultimately, improve the overall performance.

A. Contribution

In this paper, we present a novel direct GPU back-end to the FastFlow framework which allows any programmer to efficiently introduce GPU support for demanding parallel skeletal applications in heterogeneous CPU/GPU environments. Our implementation consists of OpenCL code inside the FastFlow skeletons along with static task allocation techniques in order to seamlessly launch the OpenCL back-end on the combination of GPU and CPU based on the underlying system architecture.

To evaluate the feasibility of our approach, we have implemented an image convolution algorithm as a streaming pipeline using the FastFlow framework in order to coordinate dynamic work distribution on multi-core CPU and GPU resources. Our results show significant performance improvements, typically of an order of magnitude, to previous other skeletal implementations.

In order to avoid a system crash for the large-scale problems with intensive workload size, the memory management approach provided in [7] has been used to regulate the memory footprint of the application and maintain usage within configurable bounds. It effectively complements our initial adaptive pipeline mechanisms applied to major computational problems [8].

This paper is organised as follows. In section II, we provide the background to our work along with a brief description of OpenCL and FastFlow. In section III, we explain the proposed approach for OpenCL-based skeleton and static allocation techniques inside FastFlow.In section IV we provide a test-bed to evaluate our proposed skeletons. Section V presents the experimental infrastructure used for evaluating the results, followed by the result of our evaluation. Finally, section VI provides some concluding remarks and future work.

II. Background

Large-scale, resource-heavy parallel computing applications usually have coarse-grained workloads that may only be feasibly executed on dedicated (potentially distributed) heterogeneous CPU/GPU architectures. During execution, these applications should spin off a combination of distinguishable sub-processes to be executed either on a CPU or on a GPU. They often require techniques to i) optimally coordinate the utilisation of heterogeneous resources dynamically; and ii) enable the synchronisation and communication between threads/processes in the heterogeneous environment.

As skeletons abstract commonly-used patterns of parallel computation, communication, and interaction, they arguably ought to facilitate automatic hardware-software mapping. Several skeletal libraries have furnished GPU support within their skeleton set. Bones [9] translates skeletal C code into CUDA and SPOC [10] incorporates CPU-to-GPU data transfer functionality for CUDA and OpenCL into its OCaml skeletal set. More relevant to this work are SkePU [11], SkelCL [12], and Muesli (with GPU extensions) [13], which provide a series of C++ templates to deploy fine-grain data-parallel skeletons. In general, each skeleton is generic in both the type of the data to be processed and the operations to be applied to the data. Moreover, they support the functional programming approach and take a function and data as input. SkePU functions are provided in the form of macros while SkelCL accept an input function in the form of string. Hence, SkelCL provides more flexibility in generating the input function. Multiple back-end are supported by both SkePU and SkelCL, meaning that the generated code can run on both OpenMP and OpenCL environments. Muesli is probably the most stable of the three libraries as it has been tested for a number of years. However, only one of the back-ends can be used at the runtime and as such, they are not dealing with heterogeneous environment.

OpenCL

Increasingly used for programming heterogeneous architectures composed of CPUs and GPUs or other accelerators, OpenCL distinguishes between a host system—usually containing one or several CPUs—and the devices that are targeted. An OpenCL device has logically got one or more computing units (CUs) where each CU is divided into one or more processing elements (PEs). OpenCL applications run on the host and call the kernel functions which are executed simultaneously by multiple PEs on one or more devices. A single instance of a kernel function is called a work-item and can be identified by its global identifier ('id'). Every work-item executes the same code, but the execution can vary per work-item due to branching according to its id. The work-items are organised in work groups. The arrangement of work-items within one work-group has typically got a significant effect on the overall runtime performance.

As the host and the device have separate memories in OpenCL programs, specific functions are provided to transfer data from the host memory to the device memory and viceversa. Memory areas have to be allocated on the device before data can be accessed by it and explicitly deallocated thereafter. In fact, the OpenCL kernel functions are compiled at runtime, and the host program passes the kernel source code as a plain string to the OpenCL driver in order to create executable binary code.

The variety of hardware supporting OpenCL and the runtime compilation provides portability across a wide range of devices. It allows the programmer to dynamically choose the available hardware resources and use the tuning techniques to automatically *map, and potentially dynamically remap,*

software components to the underlying hardware in order to achieve high performance results.

FastFlow

In contrast to fine-grained data parallel skeletal libraries, FastFlow provides coarse-grained stream-parallel pattern skeletons. It is a parallel programming framework for multi-core platforms which facilitates the development of shared memory parallel applications by providing abstraction layers, programming constructs, and composable algorithmic skeletons [5].

FastFlow provides three layers of abstraction between the underlying multicore/manycore hardware and the application:

- *Simple Streaming Networks* which are basically lock-free Single-Producer-Single-Consumer (SPSC) queues;
- *Arbitrary Streaming Networks* which are generalised Single-Producer-Single-Consumer (SPSC) queues that provide one-to-many (SPMC), many-to-one (MPSC), and many-to-many (MPMC) queues. Data flow and synchronisation between queues are transparently handled; and,
- *Streaming Network Patterns* which are in essence algorithmic skeletons, such as farm and pipeline—two skeletons widely-used on distributed architectures [14]—which can be arbitrarily nested or sequential combined.

FastFlow streaming patterns are coordinating mechanisms that control the flow of work between multiple concurrent threads. Such flow control allows programmers to focus on the application-specific computations by efficiently abstracting the complex coordination and communication layers. It should arguably be particularly useful in the context of heterogeneous multi-core CPU and GPU platforms.

In theory, the ability to incorporate arbitrary functions in FastFlow allows the use of GPU resources with existing kernels. Practically, GPUs present additional challenges because of their variation of capabilities, performance, and efficiency, which typically require direct intervention and tuning from the FastFlow programmer. A particularly promising prospect should therefore be the availability of a direct GPU back-end in the FastFlow framework that transparently provides hybrid CPU/GPU execution of certain kernels using a platform-neutral runtime such as OpenCL.

> The novelty of our approach lies in the addition of an OpenCL back-end to the FastFlow framework, which i) provides seamless coordination over hybrid CPU/GPU environments; and ii) allows a parallel programmer to build demanding skeletal applications with large-scale coarse-grained workloads, that may only be practically deployed on dedicated, potentially distributed, heterogeneous CPU/GPU architectures.

III. Implementation

Currently, FastFlow only provides coordination capabilities through a pipeline or a farm in multi-core CPU environments. A generic pipeline stage or a farm worker is a derived class from the abstract class called ff_node. The ff_node contains all the information required for synchronisation and communication between threads/processes. It is provided with three virtual overridable member functions:

- svc_init for initialisation, which runs once in application life time;
- svc to perform the actual computations, which runs over each unit of input data; and,
- svc_end for finalisation and cleanup, which runs once in application life time.

As a pipeline and a farm are derived from ff_node, any nested combinations of skeletons are possible. Figures 2 and 3 show the structure of the heterogeneous farm and pipeline components respectively.

We have added an OpenCL back-end to FastFlow in order to execute parallel skeletal programs on hybrid GPU/CPU environments. The OpenCL back-end is composed of three components:

- *OpenCL node*: addresses the OpenCL code inside FastFlow skeleton;
- *Task allocator*: automatically allocates an OpenCL stage/worker to available underlying architectural resources; and
- *Decision controller*: manually controls the device allocation procedure.

In the following subsections we describe each component in detail.

A. OpenCL node

An abstract class, called ff_oclNode, is derived from ff_node with two extra virtual overridable member functions:

- svc_SetUpOclObjects for creating all required OpenCL objects and building the program. It is invoked each time an underlying device changes; and,
- svc_releaseOclObjects to clean up all memory allocated to OpenCL objects. It is called each time an underlying device changes.

Any class derived from ff_oclNode may represent either a generic pipeline stage or a worker in a farm which contains OpenCL code. In order to achieve high performance and maximum device utilisation, there should be a 1-to-1 correspondence between an OpenCL kernel and the derived class.

The difference between the svc_init and the svc_end with their corresponding svc_SetUpOclObjects and svc_releaseOclObjects counterparts is the number of times they are called. Ergo, the need to be distinguished from each other. Separating the OpenCL svc_SetUpOclObjects

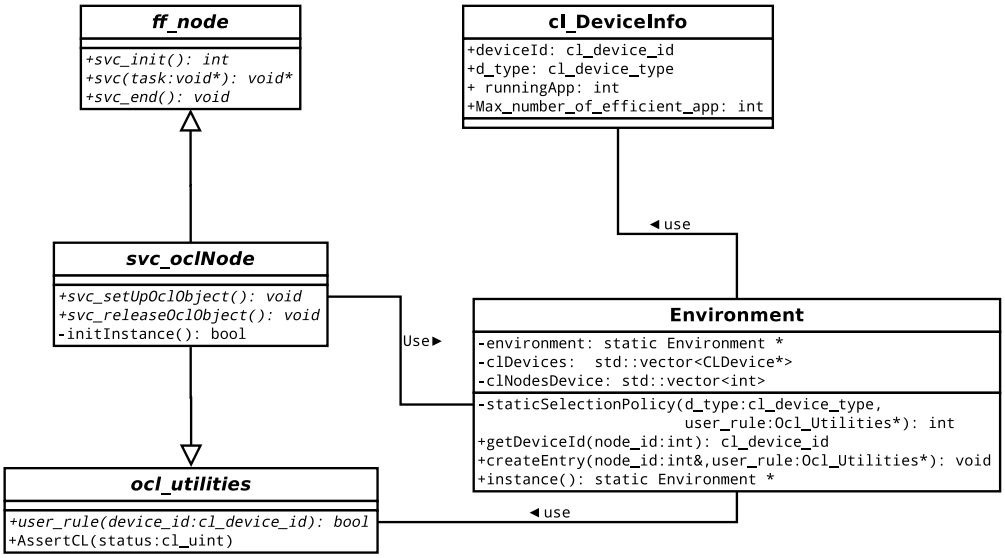

Figure 1. UML diagram for the architecture of our OpenCL-based back-end for FastFlow.

from kernel computation enables the system to build the kernel once and on each unit of input data as long as the underling device has not changed. This improves the performance as compiling and building the OpenCL kernel each time from the source is a time consuming task and can create an overhead in the system [12]. That is to say, once the object is set, they will be used as long as the same kernel for the same device is called.

B. Task allocator

A singleton class, called environment is provided with three major functionalities:

1) *Global view over the underlying OpenCL capable device*. It contains the information such as device Id , device type, number of running applications on device and maximum number of applications which can be run at the same time on each device without dropping the performance. A heuristic method is provided to calculate the maximum number of kernels per device by considering important measures such as the device type, maximum number of computing units and maximum size of memory;
2) *Allocation of each OpenCL kernel to the best appropriate device*. By considering the optimal hardware utilisation and high performance results, it provides a virtualisation layer over the underlying hardware which hides the device allocation mechanism from the user. For each kernel, a device with minimum value of rn_d/mx_d which satisfies the selection policy is selected. rn_d is the number of kernels currently running on the device and mx_d is the maximum number of kernels that can run per device. The default selection policy, tries to select the GPU device if it is available otherwise it will fall-back to CPU; and,
3) *Generation of a look-up table*. Containing one entry per kernel, the table shows the device information running on a given kernel. It provides fast device access for running the OpenCL kernel over each input unit of data, and contains the address of the device used for each node. This capability has been added for future dynamic mapping, as it monitors all available device status and dynamically reallocates the OpenCL kernel to other available devices in order to increase the performance.

The third one can eventually introduce fault tolerance in case of device crash. As a heuristic method is used to provide the best available device for each kernel, other important measures such as the type of device, maximum number of compute unit, maximum size of memory, and the kernel number can arguably be used to relocate the computation in case of device failure.

C. Decision controller

A class called oclUtilities with a virtual overridable function, called user_rule is provided to allow the user to manually control the device selection policy for codes that need special devices to run. For each kernel the user has the option to provide any OpenCL-based condition or rule needed the allocation of a kernel to a device. By default, it returns true which allows the system to use its default selection policy. The ff_oclNode is a subclass of oclUtilities and allows the user to implement user_rule for each worker or stage in the farm or pipeline respectively.

Figure 1 shows the UML diagrammatic representation of

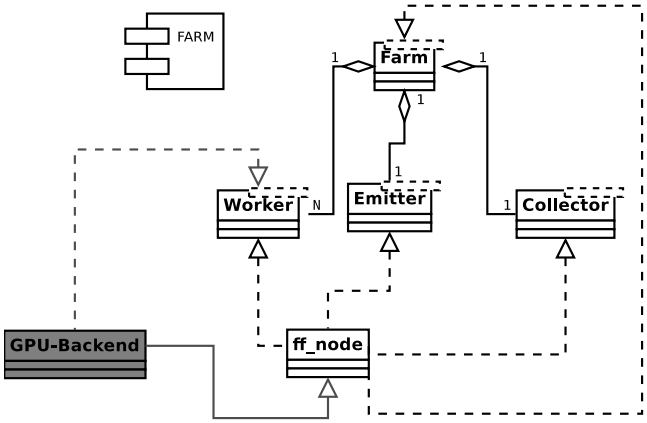

Figure 2. Heterogeneous Farm architecture in FastFlow.

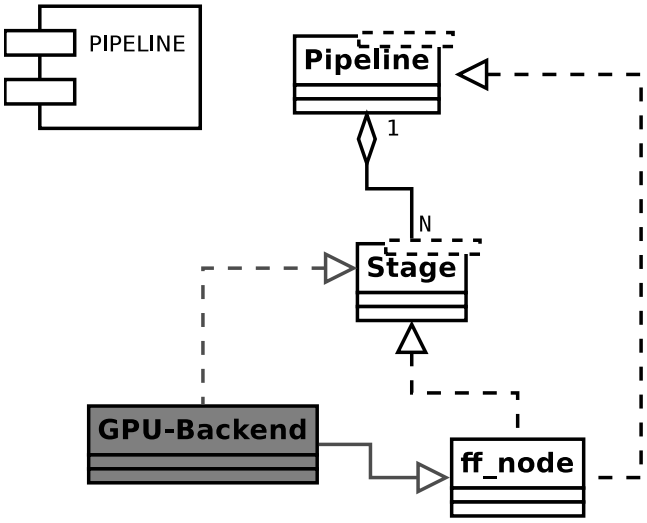

Figure 3. Heterogeneous Pipeline architecture in FastFlow.

the overall heterogeneous skeletal structure for our GPU back-end in FastFlow.

IV. Case Study

We have selected the convolution problem [15] which is widely used in image processing applications such as blur, smooth effect, or edge detection. In our case, a convolution filter is just a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels, as expressed in Equation (1).

$$output\_pixel(i,j) = \sum_m \sum_n input\_pixel(i-n, j-m) \times filter\_weight(n,m) \quad (1)$$

It is noted that the convolution problem has been previously addressed on GPU architectures for specific application domains [16]. It is noted that our intention is to convey

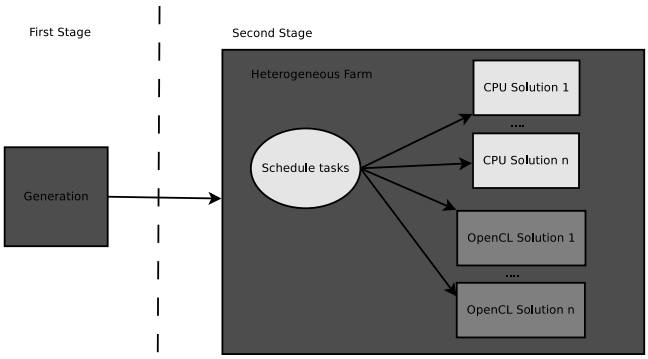

Figure 4. Visualisation of Nested Heterogeneous Pipeline.

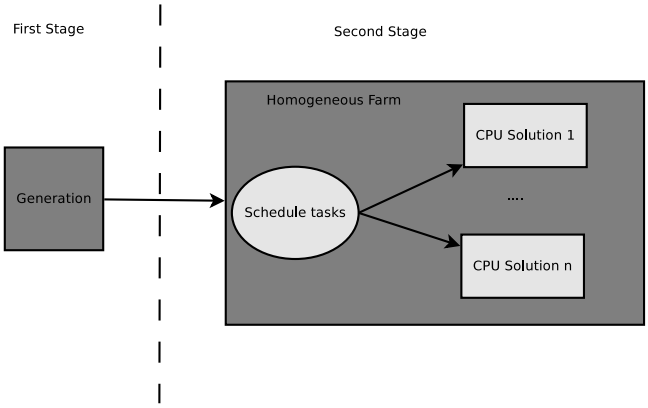

Figure 5. Visualisation of Nested Homogeneous Pipeline.

the generality of the skeletal approach on FastFlow using convolution with a computationally-demanding illustrative problem, rather than deploying the specific convolution application. In that spirit, we do not intend to manually tuned the algorithm for CPU and GPU. The whole objective of the paper is to provide seamless optimal coordination of heterogeneous nodes on hybrid architectures.

To cover the full functionality of the provided skeleton, five combinations of FastFlow skeletons have been used to implement the problem, each of which is composed of two or more of the following nodes:

1) *Generation* of suitable input matrix ($input\_pixel$) and filter weights matrix ($filter\_weight$);
2) *OpenCL_solution* using the OpenCL kernels to convolve the filter weights matrix with input matrix ($output\_pixel(i,j)$); and,
3) *CPU_solution* using the C++ code to convolve the filter weights matrix with input matrix ($output\_pixel(i,j)$).

The solution methods are as follows:

1) *Nested heterogeneous pipeline*: a nested 2-stage pipeline containing 'Generation' as the first stage and a farm with heterogeneous CPU_solution and

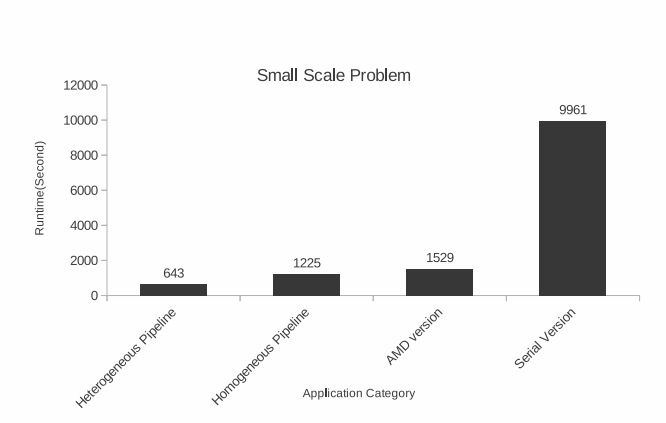

Figure 6. Total runtime execution for the small-scale convolution problem.

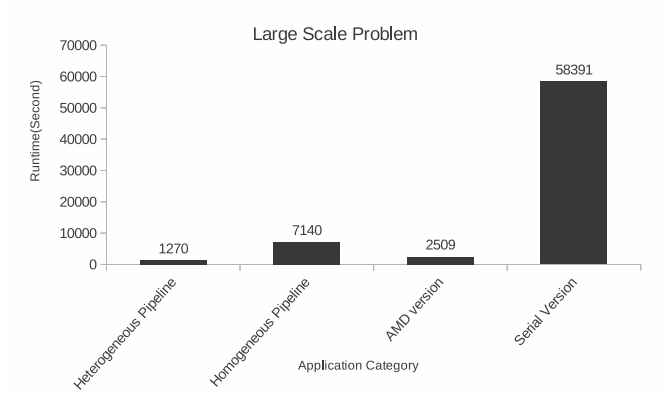

Figure 7. Total runtime execution for the large-scale convolution problem.

Table I
PROBLEM CATEGORIES

Category	Input Matrix Size	Mask Matrix	Input Matrix No.
Small Scale	2048*2048	15*15	1000
Large Scale	8192*8192	48*48	1000

OpenCL_solution workers as the second stage. Figure 5 shows the schematic view of the corresponding pipeline.

2) *Nested Homogeneous CPU pipeline*: a farm with homogeneous CPU workers. Each worker is a nested 2-stage pipeline containing Generation as the first stage and a farm with homogeneous CPU_solution workers as the second stage. Figure 4 illustrates the schematic view of the corresponding pipeline.

We have created the subclass ff_node to implement a ff_generate to represent the 'Generation' and ff_cpu_solve to represent the 'CPU_solution'. Also, the ff_gpu_solve is derived from the ff_oclNode to represent the 'OpenCL_solution.' The implementation of ff_gpu_solve is provided as an appendix at the end of this paper.

Each generated node is added to a farm or pipeline container object depending on the type of architecture used. Execution is started by an invocation to the run_and_wait_end method. As an example the implementation *Nested heterogeneous pipeline* is provided in the final appendix.

V. PERFORMANCE EVALUATION

The performance evaluation has been carried out on a machine with a hybrid CPU/GPU architecture as specified in Table II. The provided skeletons have been applied over different input size, called small scale and large scale problem. These two different classes of problem size are shown in Table I.

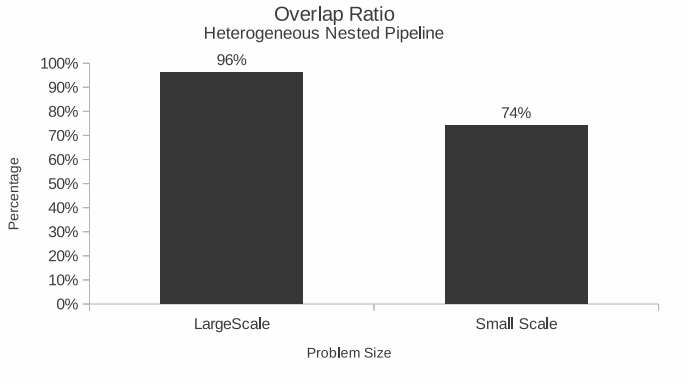

Figure 8. The ratio of GPU run-time to total run-time.

For each size of the problem, the runtime execution is compared with that of the convolution problem provided by AMD [17] as part of the standard software stack distribution as well as the serial version of the code. For all the algorithms with OpenCL nodes, the AMD OpenCL kernel implementation is used in order to have a fair comparison.

Figure 6 and 7 indicate that the heterogeneous FastFlow version is 2 and 7 times faster than the homogeneous FastFlow version respectively.

Table II
HARDWARE SPECIFICATION.

Parameter	Value
No. of CPUs	2
Cores per CPU	6
CPU Clock	3.07 GHz
Physical Memory	50 GB
No. of GPUs	1
GPU Model	NVIDIA Tesla M2090
GPU Memory	6 GB
GPU Cores	512
CUDA Version	4.0 V0.2.1221
GPU Driver Version	290.10

Table III
THE MOST APPROPRIATE NUMBER OF OPENCL_SOLUTION NODE AND CPU_SOLUTION NODE FOR EACH TEST-BED OVER DIFFERENT PROBLEM SIZE.(KEY: N/A:NOT APPLICABLE)

Test-Bed	Problem Category	CPU_solution node	OpenCL_solution node
Heterogeneous Nested pipeline	Small scale	1	4
Heterogeneous Nested pipeline	Large scale	0	1
Homogeneous Nested pipeline	Large and Small scale	8	0
AMD version	Large and Small scale	0	1
Serial version	Large and Small scale	1	0

In the small-scale category, a single OpenCL_solution worker runtime is around 6 times faster than that of CPU_solution one. As indicated in Table III, in the 'Heterogeneous Nested Farm' for the small-scale category problem 4 OpenCL_solution workers and one CPU_solution worker have been defined. However, any different combinations of CPU_solution and OpenCL_solution nodes can affect the performance of the system. The reason is that the default farm load balancer in FastFlow uses the round-robin technique to choose a node and subsequently assign the task to the node queue. This means that for an equal number of CPU/GPU workers, it will receive equal amount of workload. Therefore, the more CPU_solution nodes we have, the more portion of tasks are allocated to the CPU with, consequently, a sharper decrease in program performance. In our case for the small scale problem $1/5$ of the total tasks are allocated to the CPU_solution worker. This portion is almost equal to the ratio of OpenCL_solution runtime to OpenCL_solution runtime. In this case the selected combination of CPU/GPU workers was the optimal combination.

In the large-scale problem the situation is different. Here, there is a drastic difference between the CPU_solution nodes and the OpenCL_solution nodes, the GPU one being 50 times faster than the CPU one (see Figure 7). Having left untouched the default scheduling policy, random combinations of CPU/OpenCL_solution nodes can considerably decrease the performance, *unless* the ratio of $OpenCL\_solutionnode/CPU\_solutionnode$ is high.

Nonetheless, because of the limitation of GPU devices and the large-scale nature of the input data, our initial attempts indicated that having more than one OpenCL_solution node on one GPU at the same time can considerably decrease the performance and, even, cause the system to crash. Hence, in this case the best combination is to have one OpenCL_solution node and zero CPU_solution node. Therefore, computing the desirable combination of CPU/OpenCL_solution node is not a trivial job and can affect the performance considerably.

Moreover, in contrast to the AMD version, using FastFlow allows to parallelise not only the matrix generation and computation stage for the convolution, but also the computation stage over a heterogeneous platform. Through this possibility, our heterogeneous FastFlow version is around 3 times faster than the AMD version for small scale problems and 2 times faster than that of AMD for large scale problems. Therefore, parallelising the application and having the heterogeneous coordination over the hybrid CPU/GPU environment significantly improves the performance.

Figure 8 indicates that the total application runtime for the heterogeneous version is almost the same as the GPU run-time. This represents a good overlap of the CPU and GPU nodes in the heterogeneous skeletons, validating the choice of this architectural style for maximising resource utilisation. For the small-scale problem we have needed 4 OpenCL_solution nodes to achieve acceptable GPU utilisation while for large-scale problem by having just 1 OpenCL_solution node, the GPU utilisation has been fulfilled (see Figure 8). Therefore, as the workload of kernel on the GPU increases, the number of kernels which can run at the same time on the GPU should decrease in order to scale up the speed of algorithm or, at least, prevent the GPU performance from dropping.

It should also be noted that as the code is written in a platform-neutral language such as OpenCL, it can only run on OpenCL-capable platforms. Ergo, we can have different instances of the OpenCL_solution node and run them over different OpenCL capable devices. However, depending on the type of devices, the execution time over distinct devices can vary significantly. For example, running the OpenCL code over GPU in our case has almost been 4 times faster than running it over CPU. Hence, as long as there is no decrease in the GPU performance, all OpenCL nodes should be assigned to the GPU and, subsequently, the OpenCL node should be assigned to the CPU. Thus, finding the maximum limitation of each OpenCL capable device is important for having the maximum utilisation. Also, the optimal allocation of nodes over available devices is necessary to reach the maximum speed-up of the system.

In summary, the effective programming of heterogeneous architectures is crucial for the correct optimisation of throughput and resource utilisation. It also improves the scalability of the framework specially for large scale problems. Unsurprisingly, Figure 7 shows that the homogeneous version of FastFlow is 8 times faster than the serial version, but 3 times slower than the AMD version and 7 times slower than the heterogeneous version.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have provided heterogeneous skeletons for FastFlow to seamlessly coordinate OpenCL kernels from large-scale computational problems over hybrid CPU/GPU architectures. While demonstrating substantial and stable

scalability, the obtained FastFlow results attests to the generality of the FastFlow design and the strengths of the algorithmic skeleton approach. However, it is clear that having heterogeneous skeletons for FastFlow should be *sine qua non* for skeletal frameworks in order to increase throughput for coarse-grained and resource-intensive workloads.

Nonetheless, there are limitations in our approach. Currently, the default scheduling policy for allocating tasks to the workers in the farm is round robin. This cannot be useful when there are different workers with different run-time speeds. Although FastFlow allows user to write their own load balancer, finding an optimal portion of tasks for each worker is not a trivial job. It is foreseen that dynamic tuning should eventually be handled at runtime and through a automated heuristic mechanism. In the future, we intend to provide a run-time system with the ability to dynamically auto-tune the load balancer for arbitrary heterogenous architectures.

Moreover, checking the device status periodically can provide useful information to 'offload' the OpenCL code to the first available accelerator/device. The required information about the workload size of the OpenCL kernel and dynamic device status can be obtained at run-time. At the moment, the coordination mechanism for finding the maximum running kernels and kernel allocation over heterogeneous environment is static. Providing dynamic allocation techniques which try to monitor the device status, kernel information, and optimally assign the nodes to the best available devices is a crucial goal for our future work.

Additionally, the full support to geographically-distributed architectures in FastFlow is necessary to achieve execution on a decoupled clusters and potentially clouds. One aspect of the appeal of the skeleton concept is that the developer can potentially be insulated from the intricacies of specific libraries and their particular idiosyncrasies. A distributed version of FastFlow in under development as a part of our EC-funded ParaPhrase project.

In terms of distributed systems, when having multiple GPU kernels distributed across multiple locations, assigning an OpenCL task to the nearest available GPU should be the first option. The closest GPU is not necessarily the most suitable, since the decision of either allocating the OpenCL node to that GPU or to the local CPU should take also into account different criteria such as the workload size of the kernel and the migration cost. Furthermore, while having different OpenCL kernels, those with the largest resource-intensive workload should be assigned to a GPU and the smaller ones should fall-back to the GPU, when the maximum GPU limit for running the kernels is reached. Consequently, we aim to extend the GPU back-end to the distributed version of FastFlow.

ACKNOWLEDGMENTS

This work has been funded by the European Commission FP7 through the project *ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems*, under contract no.: 288570 (10/2011-9/2014). http://paraphrase-ict.eu

Authors' addresses: M. Goli, IDEAS Research Institute, Robert Gordon University, Aberdeen AB25 1HG, United Kingdom. http://www.rgu.ac.uk/ideas
H. González–Vélez, Cloud Competency Centre, National College of Ireland, Mayor Street, IFSC Dublin 1, Republic of Ireland. http://www.ncirl.ie/cloud

REFERENCES

[1] J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[2] A. Munshi, "The OpenCL specification," Khronos OpenCL Working Group, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, Standard Version: 1.2, Document Rev: 15, Nov. 2011.

[3] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, ser. Research Monographs in Parallel and Distributed Computing. London: MIT Press/Pitman, 1989.

[4] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers," *Software–Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating code on multi-cores with FastFlow," in *Euro-Par 2011*, ser. LNCS, vol. 6853. Bordeaux: Springer, Aug. 2011, pp. 170–181.

[6] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on multi-core with FastFlow," in *PDP 2010*. Pisa: IEEE, Feb. 2010, pp. 195–199.

[7] M. Goli, M. T. Garba, and H. González-Vélez, "Streaming dynamic coarse-grained CPU/GPU workloads with heterogeneous pipelines in FastFlow," in *HPCC-ICESS 2012*. IEEE, Jun. 2012, pp. 445–452.

[8] M. T. Garba and H. González-Vélez, "Asymptotic peak utilisation in heterogeneous parallel CPU/GPU pipelines: a decentralised queue monitoring strategy," *Parallel Processing Letters*, vol. 22, no. 2, p. 1240008, 2012.

[9] C. Nugteren and H. Corporaal, "Introducing 'Bones': a parallelizing source-to-source compiler based on algorithmic skeletons," in *GPGPU-5*. London: ACM, 2012, pp. 1–10.

[10] M. Bourgoin, E. Chailloux, and J. L. Lamotte, "Spoc: GPGPU programming through stream processing with OCaml," *Parallel Processing Letters*, vol. 22, no. 2, p. 1240007, 2012.

[11] J. Enmyren and C. W. Kessler, "SkePU: a multi-backend skeleton programming library for multi-GPU systems," in *HLPP '10*. Baltimore: ACM, Sep. 2010, pp. 5–14.

[12] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL: A portable skeleton library for high-level GPU programming," in *IPDPS-11 (workshops)*. Anchorage: IEEE, May 2011, pp. 1176–1182.

[13] S. Ernsting and H. Kuchen, "Algorithmic skeletons for multi-core, multi-GPU systems and clusters," *International Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 129–138, 2012.

[14] H. González-Vélez and M. Cole, "Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 15, pp. 2073–2094, 2010.

[15] V. Sobolev, "Convolution of functions," in *Encyclopedia of Mathematics*, M. Hazewinkel, Ed. Springer, 1994, ISBN: 978-1556080104.

[16] O. Fialka and M. Cadik, "FFT and convolution performance in image filtering on GPU," in *IV 2010*. London: IEEE, Jul. 2006, pp. 609–614.

[17] "AMD APP SDK v2.7 with OpenCL 1.2," AMD Corporation, http://developer.amd.com/, Web page, 2012, (Last Accessed: 17 Jun 2012).

APPENDIX A. CODE EXCERPTS

A. *ff_gpu_solve Implementation*

```

  int svc_init(){return 0;}

  void svc_SetUpOclObjects(cl_device_id dId){
      ...
     /* creating the context*/
     context = clCreateContext(NULL,1,&dId,NULL,
         NULL,&status);
     /* create a CL program using the kernel
         source */
     program = clCreateProgramWithSource(...);
     status = clBuildProgram(program,1,&dId,NULL,
         NULL,NULL);
     /*creating command quue*/
     commandQueue = clCreateCommandQueue(context,
         dId, prop, &status);
     /*creating buffers*/
     inputBuffer = clCreateBuffer(...);
     ...
     /* creating kernel*/
     kernel = clCreateKernel(...);
     /*** Set appropriate arguments to the kernel
         ***/
     status = clSetKernelArg(...);
     ...
     /* Check group size against
         kernelWorkGroupSize */
     status = clGetKernelWorkGroupInfo(...);

  }
  bool  device_rules(cl_device_id dId){ return
      true;}

  void * svc(void * task) {
     ...
     task_t *t =(task_t*)task;
     //writing input array A  and sub mask to the
         device buffers
     status = clEnqueueWriteBuffer(...t->input
         ...);
     ...
     status = clEnqueueNDRangeKernel(...);
     status = clEnqueueReadBuffer(...t->outpt...)
         ;
     free(t);
     popsignal();
     return GO_ON;
    }
  }

  void svc_releaseOclObjects(){
     clReleaseKernel(kernel);
     clReleaseProgram(program);
     clReleaseCommandQueue(commandQueue);
     clReleaseMemObject(inputBuffer);
     clReleaseMemObject(maskBuffer);
     clReleaseMemObject(outputBuffer);
     clReleaseContext(context);
  }

  void  svc_end() {}
};
```

c

B. *Nested Heterogeneous Pipeline*

```
int main(int argc, char * argv[]) {
  ...
  ff_pipeline pipe;

  /*nested farm within first stage of pipeline*/
  ff_farm<> farm1;
  Collector C;
  farm1.add_collector(&C);
  std::vector<ff_node *> w1;
  for(int i=0;i<nworkers;++i)
  w1.push_back(new Generate_Stage);
  farm1.add_workers(w1);

  /*Add stages to pipeline*/
  pipe.add_stage(&farm1);

  /*nested farm within second stage of pipeline*/
  ff_farm<> farm;
  farm.add_collector(NULL);
  std::vector<ff_node *> w;
  for(int i=0;i<gnworkers;++i)
  w.push_back(new Solve_Stage);
  farm.add_workers(w);

  /*Add stages to pipeline*/
  pipe.add_stage(&farm);

  ...
  /* start the pipeline*/
  pipe.run_and_wait_end()

}
```